TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

ÔN TẬP
Kiến trúc máy tính - CO2008

Binh Tran-Thanh
thanhbinh.hcmut@gmail.com
Khoa Khoa học và Kỹ thuật Máy tính
Đại học Bách Khoa Tp.HCM

December, 2020

Nội dung

1 Kiểu lệnh

R-type

I-type

J-type

- Op (opcode) Mã lệnh, dùng để xác định lệnh thực thi (đối với kiểu R, Op = 0).
- Rs, Rt, Rd (register): Trường xác định thanh ghi (5-bit). vd: Rs = 4 có nghĩa là Rs đang dùng thanh ghi a0 hay thanh ghi 4.
- Shamt (shift amount): Xác định số bits dịch trong các lệnh dịch bit.
- Function: Xác định toán tử (operator hay còn gọi là lệnh) trong kiểu lệnh R.
- Immediate: Đại diện cho con số trực tiếp, địa chỉ, offset.

1.1 Bài tập

Bài 1.1. Cho đoạn code hợp ngữ MIPS sau:

```
        addi $a0, $zero, 100    # upper threshold
        addi $a1, $zero, 0      # count variable
        add  $a2, $zero, $zero  # sum initialization
loop:
        beq  $a0, $a1, exit
        add  $a2, $a2, $a1
        addi $a1, $a1, 1
        j    loop
exit:
```

(a) Xác định loại lệnh của từng lệnh trong đoạn code trên.

(b) Xét lệnh `beq $a0, $a1, exit`, xác định khoảng cách rẽ nhánh tối đa (đơn vị tính theo byte, theo lệnh) của lệnh beq

(c) Giả sử địa chỉ bắt đầu của đoạn chương trình trên là `0x1000_0000`. Xác định mã máy của lệnh `j loop`

(d) Giả sử lệnh `j loop` có mã máy là `0x0809_0A0B`, và PC hiện tại là `0x4000_0080`. Sau khi thực thi lệnh `j loop` giá trị thanh PC là bao nhiêu.

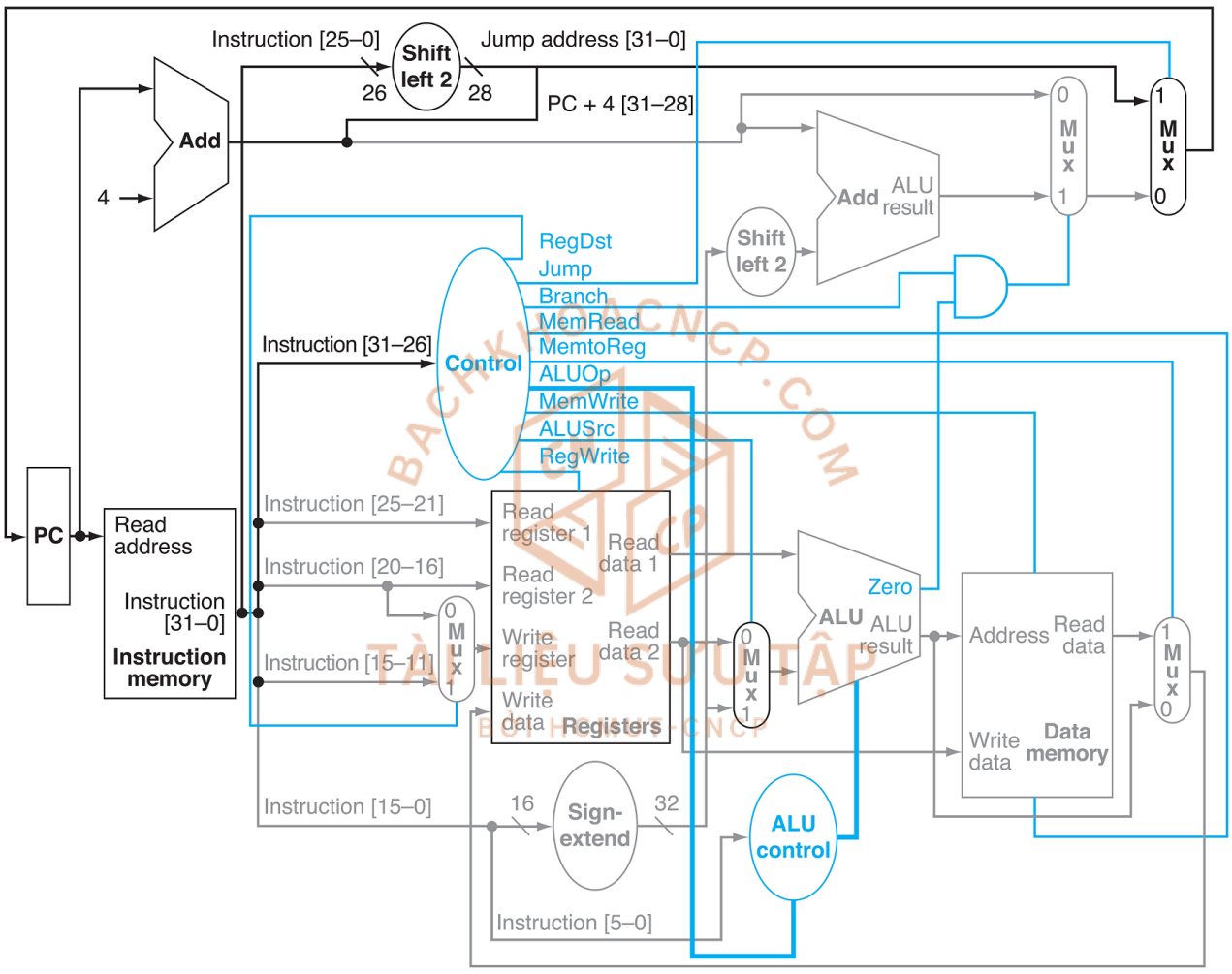
2 Single clock processor

2.1 Single clock processor

- Ưu điểm: Một lệnh thực thi trong một chu kỳ. Máy tính đơn giản, dễ hiểu, dễ hiện thực.
- Nhược điểm: Một chu kỳ tốn nhiều thời gian, mỗi lệnh dù nhanh hay chậm đều thực thi trong một chu kỳ → hiệu suất thấp.

2.2 Kiến trúc single clock cycle

2.2.1 Kiến trúc single clock cycle



Hình. 1: Kiến trúc bộ xử lý MIPS single clock cycle

- Thanh PC: Trỏ đến lệnh thực thi kế tiếp.
- Instruction memory: Chứa code, chương trình thực thi (phần .text).
- Registers file: Gồm 32 thanh ghi, do đó cần 5 bit để xác định thanh ghi (2^5 = 32). Để xác định chi tiết thanh ghi, ta tham khảo Bảng 1.
- Sign-extend: Bộ mở rộng dấu, mở rộng dấu 16 bits → 32 bits.
- Bộ chọn (MUX): Dùng để chọn input cho output tương ứng. Tín hiệu select đưa ra sự lựa chọn.
- ALU: Thực hiện tính toán.
- Data memory: Là vùng nhớ dữ liệu (phần .data). Chỉ có lệnh LOAD và STORE mới truy xuất vào khối Data Memory.

- Control: Khối điều khiển, sinh ra tín hiệu điều khiển dựa vào mã lệnh Opcode.

Bảng. 1: Danh sách 32 thanh ghi

REGISTERS		
0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

Bảng. 2: Ý nghĩa của các tín hiệu điều khiển.

Ta giả sử tín hiệu tích cực ở mức cao (HIGH).

Tín hiệu	**Ý nghĩa**	**Tích cực (1)**	**KHÔNG tích cực (0)**
RegDest	Chọn thanh ghi kết quả	Chọn R_d để ghi kết quả	Chọn R_t để ghi kết quả
RegWrite	Ghi kết quả vào thanh ghi	Cho phép ghi	Không cho phép
ALUSrc	Chọn toán hạng cho ALU:thanh ghi hoặc số (immediate)	Chọn số	Chọn thanh ghi
Memwrite	Ghi vào data memory	Cho phép ghi	Không cho phép
MemRead	Đọc từ data memory	Cho phép đọc	Không cho phép
MemtoReg	Dùng để chọn đường từ data memory đến thanh ghi	Dữ liệu data memory → thanh ghi (lệnh load)	Kết quả ALU → thanh ghi
Branch	Dùng cho lệnh rẽ nhánh có điều kiện	Lệnh branch	Không phải lệnh branch
Jump	Dùng cho lệnh nhảy không có điều kiện	Lệnh jump	Không phải là lệnh jump

Chú ý: Không quan tâm đến RegDst, MemtoReg khi tín hiệu RegWrite = 0

2.2.2 Các bước thực thi lệnh MIPS

Bảng. 3: Các bước thực thi của các lệnh MIPS.

ALU	Instruction Fetch	Decode	Execute	Write Back	
Load	Instruction Fetch	Decode	Execute	Memory Read	Write Back
Store	Instruction Fetch	Decode	Execute	Memory Write	
Branch	Instruction Fetch	Decode	Execute		
Jump	Instruction Fetch	Decode			

2.3 Bài tập

Dùng lại kiến trúc được miêu tả ở Hình 1 để trả lời các câu hỏi bên dưới:
Cho bảng delay của các khối phần cứng như Bảng bên dưới:

I-MEM	ADDER	MUX	ALU	REG	D-MEM	Control
200ps	10ps	30ps	180ps	150ps	200ps	10ps

Bài 2.1. Xác định đường đi có độ trễ lâu nhất của lệnh `and`, `load` và tính độ trễ đó?

Bài 2.2. Xác định các tín hiệu của khối control khi thực thi lệnh `beq $8, $9, label`. Với `$8 = 0x00FF, $9 = 0x00FE`

Bài 2.3. Thành phần phần cứng nào không sử dụng khi ta thực thi lệnh `slti`, lệnh `j`.

Bài 2.4. Bỏ qua delay của các khối adder, mux, control. Xác định data path và thời gian của các loại lệnh sau.

- ALU
- Load
- Store
- Branch
- Jump

Bài 2.5. Xác định thời gian của single cycle và multi-cycle

Bài 2.6. Giả sử có 1 chương trình gồm 40% ALU, 20% Loads, 10% Stores, 20% Branches, và 10% Iumps. Tính CPI trong trường hợp single cycle, multi cylce.

Bài 2.7. Tính speed up của hệ thống Multi cycle đối với hệ thống single clock cycle.

2.4 Đáp án/Gợi ý

Bài 2.1. Critical path.

- and: I-MEM(200) → REGs(150) → MUX(30) → ALU(180) → MUX(30) = 590
- load: I-MEM (200) → REGs(150) → ALU(180) → D-MEM(200) → MUX(30) = 760

Bài 2.2. Xác định tín hiệu điều khiển.
Xét lệnh `beq $8, $9, label`. Với `$8 = 0x00FF, $9 = 0x00FE`.

Tín hiệu	Giá trị	Giải thích
RegDest	x	don't care
RegWrite	0	Không ghi kết quả vào thanh ghi
ALUSrc	0	Chọn thanh ghi để so sánh với thanh ghi $8
Memwrite	0	Không truy xuất vào vùng data
MemRead	x	dont care
MemtoReg	x	don't care
Branch	1	Lệnh rẽ nhánh
J	0	Không phải lệnh jump
ALUop	10	Tham khảo slide

Bài 2.3. Hoạt động khối phần cứng.

Lệnh `slti` dùng để set giá trị thanh ghi đích lên 1 nếu thanh ghi đem so sánh nhỏ hơn số cho trước, ngược lại nó sẽ reset giá trị thanh ghi đích xuống 0 nếu thanh ghi đem so sánh lớn hơn hoặc bằng số cho trước.

Ví dụ: `slti $t1, $t2, 100` Khi thanh ghi `$t2 < 100` thì `$t1 = 1`, ngược lại khi `$t2 >= 100` thì `$t1 = 0`. Từ đó ta xét các khối' phần cứng mà lệnh `slti` đi qua như sau:

PC (lệnh nào cũng qua PC) → I-MEM (lệnh nào cũng qua I-MEM) → control unit, Reg files, MUX (2x), Sign extend, (không đi qua bộ cộng PC) → ALU → (không qua D-MEM) → MUX → Reg files.

Bài 2.4. :))

Bài 2.5. Thời gian chu kỳ của hệ thống single cycle và hệ thống multi cycle

- Thời gian chu kỳ của hệ thống single cycle = max(thời gian thực thi của tất cả các lệnh). Lệnh load là lệnh có thời gian thực thi lâu nhất 760 ps → thời gian chu kỳ của single cycle = 760 ps.
- Thời gian của hệ thống multi cycle = maxIF (Instruction Memory), ID (Register Files), EXE (ALU), MEM (Data Memory), WB (Register Files) = max(200, 150, 180, 200, 150)= 200 → thời gian của multi cycle = 200ps.

Bài 2.6. CPI (Cycle per instruction: số chu kỳ trên lệnh).

- CPI của hệ thống single cycle = 1 (1 chu kỳ thực thi 1 lệnh).
- CPI của multi cycle = $0.4 \times 4 + 0.2 \times 5 + 0.1 \times 4 + 0.2 \times 3 + 0.1 \times 2 = 3.8$

(ALU 4 cycles; Load 5 cycles; Store 4 cycles; Branch 3 cycles; Jump 2 cycles)

Bài 2.7. Thời gian thực thi.
Thời gian chạy của 1 chương trình = $IC \times CPI \times$ *thời gian 1 chu kỳ* = $IC \times CPI/f$.

- IC: số lệnh.
- CPI: số chu kỳ trên một lệnh.
- f: tần số.

Thời gian chạy trên hệ thống single cycle: $IC \times 1 \times 760$.

Thời gian chạy trên hệ thống multi cycle: $IC \times 3.8 \times 200$.

$Speedup = (1 \times 760)/(3.8 \times 200) = 760/760 = 1.00$.

Trong trường hợp này ta thấy hệ thống Multi clock cycle lại không hiệu quả bằng hệ thống Single clock cycle vì:

- Các bước thực thi có thời gian không đều.
- Những lệnh nhiều bước (load, store) chiếm phần lớn trong chương trình
- Trong hệ thống single clock cycle quá trình write back được thực thi ở chu kỳ tiếp theo nên cycle time của hệ thống single clock cycle giảm.

3 Pipeline processor

3.1 Pipeline processor

3.1.1 Các bước trong pipeline

Bộ xử lý Pipeline chia quá trình thực thi lệnh thành 5 bước, mỗi bước thực thi trong trong một chu kỳ.

1. **IF**: Lấy lệnh (khối Instruction Memory), 32bits lệnh chứa các thông tin của 1 lệnh được lấy ra từ instruction memory.
2. **ID**: Giải mã lệnh (khối Registers và Control), xác định toán tử, các tín hiệu điều khiển, nội dung các thanh ghi, giá trị immediate.
3. **EXE**: Thực thi tác vụ lệnh (khối ALU).
4. **MEM**: Truy xuất vùng nhớ (khối Data Memory) - chỉ dùng cho lệnh `load`/`store`.
5. **WB**: Ghi kết quả vào thanh ghi (khối Registers).

3.1.2 Hiệu suất.

Ta chia quá trình thực hiện lệnh ra thành k bước (tổng quát).

- Thời gian thực thi N lệnh trên hệ thống single cycle = N $\times$ single cycle.
- Thời gian thực thi N lệnh trên hệ thống pipeline = (k + N -1) $\times$ pipeline cycle. *Lệnh đầu tiên: mất k chu kỳ, n -1 lệnh còn lại, còn lại mỗi lệnh 1 chu kỳ.*

  Trường hợp lý tưởng: thời gian chu kỳ single cycle = k $\times$ thời gian chu kỳ pipeline cycle.

  Khi đó

$$speedup = \frac{k \times n \times pipelinecycle}{(k + n - 1) \times pipelinecycle}$$

  Khi chương trình đủ lớn $n \to \infty$ thì $speedup \to k$ (pipeline nhanh tối đa gấp k lần single cycle)

Chú ý:

- Pipeline không rút ngắn thời gian thực thi của một lệnh, mà chỉ tăng hiệu suất lên bằng cách tăng thông năng (through-put) của máy.
- Khi các bước của một lệnh có thời gian thực thi khác nhau (các bước không đều nhau) thì sẽ làm giảm speed up.
- Thời gian fill và drain cũng đồng thời làm giảm speed up.
- Để hiện thực pipeline người ta dùng thanh ghi để lưu kết quả lại ở mỗi bước.
  - Tín hiệu từ khối control unit (main control)
  - Tất cả tín hiệu điều khiển được sinh ra ở bước ID
  - Mỗi bước dùng 1 số tín hiệu điều khiển
  - RegDst được dùng trong bước ID
  - ExtOp, ALUSrc, ALUCtrl, J, Beq, Bne, zero được dùng trong bước EXE
  - MemRead, MemWrite, MemtoReg dùng trong bước MEM
  - RegWrite dùng trong bước WB

3.2 Hazard

Khi hiện thực pipeline sẽ gây ra các loại hazards: structural, data, và control.

3.2.1 Structural hazard

Xảy ra khi có sự tranh chấp tài nguyên phần cứng, 2 lệnh cùng dùng chung phần cứng trong cùng chu kỳ.

3.2.2 Data hazards

Xảy ra khi có sự phụ thuộc dữ liệu kiểu Read After Write (RAW).

Các kiểu phụ thuộc dữ liệu

- Read After Write – RAW Hazard

```
add $s1, $s2, $s3       #thanh ghi $s1 duoc ghi
sub $s4, $s1, $s3       #thanh ghi $s1 duoc doc
```

Pipeline

c1	c2	c3	c4	c5	c6
IF	ID	EXE	MEM	WB	
	IF	ID	EXE	MEM	WB

Data hazard xuất hiện khi lệnh (2) đọc nội dung $s1 ở bước ID (chu kỳ 3), trong khi đó lệnh (1) lại cập nhập kết quả của $s1 ở bước WB (chu kỳ 5).

- Write After Read: Name Dependence

```
sub $t4, $t1, $t3 # $t1 duoc doc truoc
add $t1, $t2, $t3 # $t1 duoc ghi sau
```

Không có sự phụ thuộc dữ liệu ở đây, chỉ có phụ thuộc tên biến $t1. Để loại bỏ sự phụ thuộc về tên biến, ta đổi tên thanh ghi.

```
sub $t4, $t1, $t3
add $t5, $t2, $t3
```

- Write After write: Name Dependence

```
sub $t1, $t4, $t3 # $t1 duoc ghi
add $t1, $t2, $t3 # $t1 duoc ghi lan nua
```

Không có sự phụ thuộc dữ liệu ở đây, chỉ có phụ thuộc tên biến. Kết quả chỉ phụ thuộc vào lệnh (2). Để loại bỏ sự phụ thuộc về tên biến, ta đổi tên thanh ghi.

```
sub $t1, $t4, $t3
add $t5, $t2, $t3
```

- Read After Read: không gây ra sự phụ thuộc.

3.2.3 Phương pháp giải quyết data hazard.

Chèn stall (aka bubble, delay)

Chèn stall để đảm bảo lệnh trước cập nhập kết quả trong khi lệnh sau (phụ thuộc vào kết quả đó) có thể đọc được kết quả đó trong cùng chu kỳ. Phương pháp này không tốn tài nguyên phần cứng (giá thành), chỉ tạo ra delay cho chương trình (giảm hiệu suất).

Dùng kỹ thuật forwarding (xúc tiến sớm).

- Phương pháp này cần thêm tài nguyên phần cứng (giá thành tăng) để hiện thực kết hợp với chèn stall khi cần thiết.
- Khi xảy ra data hazard đối với lệnh load và lệnh kề tiếp nó, cho dù ta có dùng kỹ thuật forward thì cũng phải tốn 1 stall để giải quyết chúng.
- Để hiện thực forward, người ta thêm bộ mux cho việc lựa chọn input cho ALU.
  Các lệnh (ngoại trừ lệnh load) thì kết quả được cho ra ở bước ALU (EXE) nên khi ta dùng kỹ thuật forward sẽ không còn stall nữa.

Chú ý:

- Các forwarding đều forward về vị trí EXE vì ALU là nơi bắt đầu tính toán → cần đưa input trước ALU.
- Forward kết quả về chu kỳ n từ chu kỳ n-1 (là chu kỳ trước n).

Sắp xếp lại code

sắp xếp lại code (phần mềm) có thể giúp giảm thiểu stalls nhưng việc sắp xếp phải đảm bảo thứ tự trước sau khi có sự phụ thuộc và đảm bảo tính đúng đắn của chương trình.

3.3 Control hazard

Xét đoạn chương trình bên dưới.

```
        beq  $t1, $t2, label
        addi $t1, $zero, 100
        addi $t2, $zero, 100
        j exit
label:  addi $t1, $zero, 10
        addi $t2, $zero, 10
exit:
```

Sau khi IF (lấy lệnh) **beq**, ở chu kỳ tiếp theo ta giải mã **beq** và lấy lệnh tiếp theo. Câu hỏi là lệnh tiếp theo sau lệnh branch là lệnh nào?

- Khi $t1 = $t2, lúc đó điều kiện lệnh beq thỏa nên nó rẽ nhánh đến label thì lệnh tiếp theo là lệnh ở dòng thứ 5.
- Khi $t1 != $t2, lúc đó điều kiện lệnh beq không thỏa nên nó thực hiện lệnh tiếp theo, khi đó lệnh tiếp theo là lệnh ở dòng thứ 2.

Hiện tượng trên là control hazard. Để giải quyết control hazard ta có thể dùng phương pháp chèn stall hoặc tiên đoán kết hợp chèn stall.
Ta biết bước EXE là bước hiện thực việc so sánh điều kiện, sau đó bước MEM sẽ cập nhập thanh ghi PC. Đó đó sau bước MEM ta mới biết chính xác là lệnh tiếp theo sau lệnh branch là lệnh nào. Nên ta cần phải chèn 3 stall (khi đó bước IF nằm sau bước MEM của lệnh branch) để loại bỏ control hazard.
Trong trường hợp có thêm bộ phần cứng để so sánh trước 3 thanh ghi ở bước ID, và tính toán địa chỉ rẽ nhánh ngay lúc đó, thì ta chỉ cần mất 1 chu kỳ (1 stall) để giải quyết control hazard.
Ngoài ra ta có thể dùng tiên đoán để tăng hiệu suất của chương trình. Tiên đoán 1 bit và tiên đoán 2 bit

3.4 Bài tập

Thời gian delay của mỗi khối cho như bảng bên dưới.

I-MEM	ALU	REG	D-MEM
200ps	150ps	200ps	200ps

Bài 3.1. Bỏ qua độ trễ của khối ADD, MUX, Control. Tính thời gian chu kỳ của hệ thống single cycle, pipeline clock?

Bài 3.2. Tính thời gian thực thi của chương trình gồm 150 line code đối với single cycle và pipeline. Từ đó tính speed up để so sánh single cycle và pipeline (không có stall)

Bài 3.3. Giả sử chương trình không có stall và thống kê được là có ALU 50%, Beq 25%, lw 15%, sw10%. Tính speed up giữa multi cycle và pipeline.

Dùng đoạn code sau để trả lời các câu hỏi bên dưới

```
addi $t1, $zero, 100
addi $t2, $zero, 100
add  $t3, $t1, $t2
lw   $t4, 0($a0)
lw   $t5, 4($a0)
and  $t6, $t4, $t5
sw   $t6, 8($a0)
```

Bài 3.4. Xác định sự phụ thuộc RAW (read after write) giữa các lệnh và thanh ghi nào gây ra sự phụ thuộc đó.

Bài 3.5. Chèn stall để giải quyết hazard trên, cần bao nhiêu stall?

Bài 3.6. Sắp xếp lại thứ tự các lệnh sao cho khi chạy đoạn code đó thì ít stall nhất mà chương trình vẫn giữ tính đúng đắn.

Bài 3.7. Dùng kỹ thuật forward để giải quyết hazard khi đó còn bao nhiêu stall.

3.5 Đáp án/gợi ý

Bài 3.1. Thời gian chu kỳ single cycle, pipeline.

Single cycle = thời gian thực thi lệnh dài nhất (lệnh load).
= I-Mem → Regs → ALU → D-Mem.
= 200 + 200 + 150 + 200 = 750ps
Pipeline clock = max (I-Mem, Regs, ALU, D-Mem, Regs) = 200ps

Bài 3.2. Thời gian thực thi chương trình, speed up giữa single cycle và pipeline.

$CPI_{singleclockcycle} = 1$.
$CPI_{pipeline} = 1$.
$Time_{singlecycle} = 150 \times 750 = 112500ps$
$Time_{Pipeline} = (5 + 150 - 1) \times 200 = 30800ps$
Speed up = $\frac{142500}{30800} = 3.65$

Bài 3.3. Thời gian thực thi chương trình, speed up giữa multi cycle và pipeline

$CPI_{MultiCycle}$ = 50%×4 + 25%×3 + 15%×5 + 10%×4= 3.9
$CPI_{Pipeline}$ = 1.
Time = Số lệnh × CPI × thời gian 1 chu kỳ.
$Time_{MultiCycle}$ = 150×3.9×200.
$Time_{Pipeline}$ = (5 + 150 - 1)×1×200.
Speed up = 3.80.

Bài 3.4. Phụ thuộc dữ liệu Read After Write

```
addi $t1, $zero, 100
addi $t2, $zero, 100
add  $t3, $t1, $t2
lw   $t4, 0($a0)
lw   $t5, 4($a0)
and  $t6, $t4, $t5
sw   $t6, 8($a0)
```

Lệnh (3) phụ thuộc lệnh (2) và (1).
Lệnh (6) phụ thuộc lệnh (5) và (4).
Lệnh (7) phụ thuộc lệnh (6).

Bài 3.5. Giải quyết data hazard bằng cách chèn stall

6 stall
2 stall giữa (2) và (3).
2 stall giữa (5) và (6).
2 stall giữa (6) và (7).
Lúc chèn stall vào đảm bảo là những chỗ cần giải mã giá trị thanh ghi (ID) phải cùng chu kỳ ghi kết quả (WB)

Bài 3.6. Sắp xếp lại lệnh.

Một trong những cách sắp xếp làm giảm stall
(4) → (5) → (1) → (2) → (6) → (3) → (7)

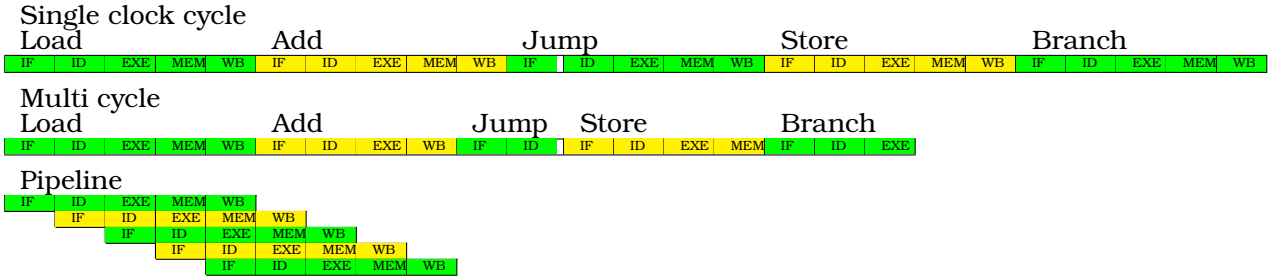
```
lw   $t4, 0($a0)      #4
lw   $t5, 4($a0)      #5
addi $t1, $zero, 100  #1
```

```
addi $t2, $zero, 100    #2
add  $t6, $t4, $t5      #6
add  $t3, $t1, $t2      #3
sw   $t6, 8($a0)        #7
```

Còn lại 1 stall giữa 6 và 3

Bài 3.7. Giải quyết data hazard bằng forwarding.
Còn 1 stall giữa lệnh (5) và (6).

Hình ảnh so sánh hệ thống single cycle, multi cycle và pipeline cycle



- **Single Clock Cycle**: Một lệnh thực thi trong 1 chu kỳ. Ví dụ lệnh load thực thi trong 1 chu kỳ(màu xanh), lệnh store thực thi trong 1 chu kỳ(màu vàng). Thời gian giữa màu xanh và vàng là bằng nhau.
- **Multi Clock Cycle**: 1 lệnh thực thi trong nhiều chu kỳ. Ví dụ lệnh Load thực thi trong 5 chu kỳ (5 chu kỳ nhỏ này tương ứng với 1 chu kỳ lớn bên single clock cycle), lệnh Store thực thi trong 4 chu kỳ (màu vàng).
- **Pipeline** Lệnh đầu tiên thực thi 5 chu kỳ, các lệnh còn lại sau mỗi chu kỳ hoàn thành xong một lệnh.

Các yếu tố ảnh hưởng đến hiệu suất của hệ thống:

- Độ dài của chương trình (instruction count)
- Số chu kỳ trên 1 lệnh (CPI)
- Thời gian của 1 chu kỳ (clock cycle time).

4 Memory

4.1 Memory

SRAM và DRAM (xem slide)

4.2 Cache

Nguồn gốc Cache: do tốc độ phát triển của CPU quá nhanh so với Memory nên khi CPU truy xuất Memory sẽ tạo ra delay khá lớn, do đó cần có cache để làm bộ đệm giữa ALU và MEM.
Cache thường được làm bằng SRAM nên nó truy xuất nhanh, và dụng lượng nhỏ giúp giảm giá thành.

Bảng. 4: Tốc độ và thời gian truy xuất của các loại memory

Type	Size	Access time
Registers	size < 1 KB	< 0.5 ns
Level 1 Cache	size 8 – 64 KB	1 ns
Level 2 Cache	512KB – 8MB	3 – 10 ns
Main Memory	4 – 16 GB	50 – 100 ns
Disk Storage	> 200 GB	5 – 10 ms

Temporal Locality: (tính cục bộ về thời gian) một biến, thực thể được truy xuất thì có thể nó sẽ được truy xuất lần nữa. Thường xuất hiện trong những vòng lặp, hay gọi hàm/thủ tục nhiều lần. Đối với truy xuất theo thời gian thì xu hướng thường giữ block đó trong cache nhằm truy xuất lại lần sau.

Spatial Locality: (tính cục bộ về không gian) lệnh/data trong vùng nhớ khi được truy xuất có thể các lệnh/data gần nó sẽ được truy xuất. Thường xuất hiện trong khai báo mảng, thực thi tuần tự. Đối với truy xuất theo không gian thì xu hướng thường chuẩn bị trước block kế tiếp.

4.2.1 Block placement

Phương pháp đặt block vào cache.

Direct mapped

Mỗi block được xác định một vị trí đặt duy nhất. Cho N là số set (= số block) trong cache. Block ID M trong bộ nhớ (RAM) sẽ được đặt vào vị trí set M % N trong cache.

Fully associative

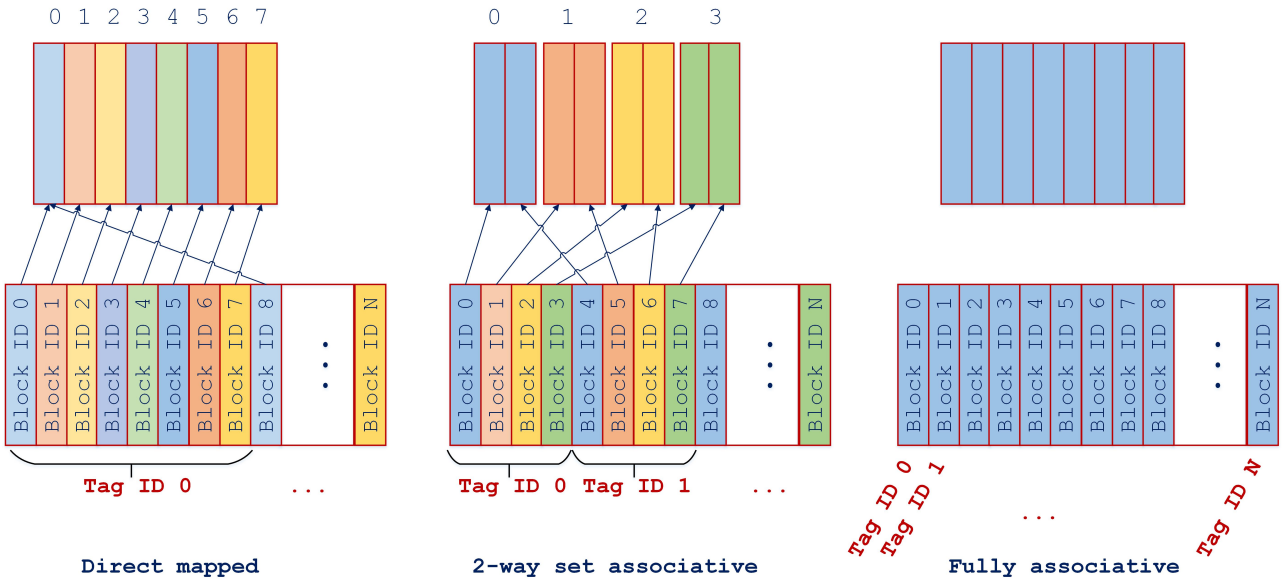
Block được đặt vào bất kỳ vị trí nào còn trống trong cache.

K-way set associative

Một set bao gồm K blocks (K có dạng 2^x). Trong set đó có K sự lựa chọn. Cho N là số set (K block tạo thành 1 set) trong cache, Block thứ M trong bộ nhớ (RAM) sẽ được đặt vào vị trí set M % N trong cache.

4.2.2 Block identification

Để xác định địa chỉ người ta chia địa chỉ ra làm 3 phần Tag, Index, block offset.



Hình. 2: Hình ảnh so sánh 3 cấu hình Direct map, k-way associative, full associative. Với k =2

Block offset

Xác định thành phần trong block được truy xuất. Để xác định block offset có bao nhiêu bit, ta đi xác định trong block đó có bao nhiêu phần tử. Xác định số phần tử bằng cách lấy (size of block)/(size of đơn vị truy xuất).
Byte-offset Xác định byte trong block.
Half-word-offset Xác định 2 bytes trong block.
Word-offset Xác định word trong block.

Index

Dùng để xác định số set trong bộ nhớ đệm.

- Direct mapped: 1 block là 1 set.
- K-way Set Associative: k block tạo thành 1 set.
- Fully Associative: toàn bộ block thành 1 set. Index lúc này là 0 bit – không cần xác định set.

Xác định số block bằng cách lấy (size of cache)/(size of block).

Tag

Để xác định block nào đang nằm trong cache hoặc kết hợp với index để xác định block ID trong RAM.
Tag bit = không gian địa chỉ – index – bytes offset.
Trong trường hợp không đề cập đến không gian địa chỉ thì ta dùng không gian 32 bit.

4.2.3 Block replacement

Khi một block vào mà không còn chỗ trống để đặt vào thì cần phải thay block cũ bằng block mới. Dưới đây là 1 số giải thuật thay thể cơ bản.

- Trong trường hợp direct mapped, vì mỗi set chỉ có 1 block nên khi nào có block mới vào thì block cũ bị thay thế do đó không có chính sách thay thế trong trường hợp này.
- FIFO: cái nào được đặt vào trước thì sẽ được lấy ra trước.
- Ramdom
- LRU: (least recently used) cái nào ít dùng nhất thì được thay thế trước.
- FILO: vào trước ra sau.

4.2.4 Write strategy

Chiến lược ghi ngược lại cache, memory

Write Through

Updata cả cache và memory, cần bit valid để xác định block đó có valid hay không.

- Đơn giản, dễ hiện thực
- Tốn lưu lượng băng thông của hệ thống vì phải update nhiều.

Write Back

Chỉ updata cache, khi có yêu cầu hay cần thay thế thì sẽ update giá trị sau cùng xuống memory. Cần bit valid để xác định block đó có valid hay không và bit modified để xác định block đó có update chưa.

- Khó hiện thực
- Ít tốn lưu lượng băng thông của hệ thống.

4.2.5 Miss/hit

- Miss: cần truy xuất mà tìm không thấy trong cache. Do đó phải đưa block chứa địa chỉ cần truy xuất vào cache, và sau đó truy xuất lại.
- Hit: cần truy xuất và tìm thấy cái muốn truy xuất trong cache.
- Ngoài dữ liệu, bộ nhớ đệm còn thêm các trường thông tin:
  - Valid: xác định có block tồn tại trong set hay không.
  - Tag: xác định block ID nào đang chứa trong block của Cache.
- Miss Penalty: số chu kỳ để xử lý cache miss.
- $Hit\_rate = \dfrac{Mit\_times}{Hit\_times + Miss\_times}$.
- $Miss\_rate = \dfrac{Misstimes}{Hit\_times + Miss\_times} = 1 - Hit\_rate$.
- I-Cache Miss Rate = Miss rate trong lúc truy xuất I-MEM.
- D-Cache Miss Rate = Miss rate trong lúc truy xuất D-MEM

Ví dụ 1: Chương trình có 1000 lệnh trong đó có 25% là load/store. Biết lúc đọc I-MEM bị miss 150, D-MEM bị miss 50. Tìm I-Cache Miss Rate, D-Cache Miss Rate.

- I-Cache Miss Rate = số lần miss instruction / số lần truy xuất I-MEM = 150/1000 = 15%.
- D-Cache Miss Rate = số lần miss data/ số lần truy xuất D-MEM = 50/(1000×25%) = 50/250 = 20%.

4.3 CPI

Khi cache miss thì sẽ gây ra stall. Để xác định bao nhiêu stall, ta đi tìm các thông số sau:

- Memory stall cycles = Combined Misses × Miss Penalty.
- Miss Penalty: số chu kỳ để giải quyết việc miss.
- Combined Misses = I-Cache Misses + D-Cache Misses.
- I-Cache Misses = I-Count × I-Cache Miss Rate.
- D-Cache Misses = LS-Count × D-Cache Miss Rate.
- LS-Count (Load & Store) = I-Count × LS Frequency.

- Memory Stall Cycles Per Instruction = Combined Misses Per Instruction × Miss Penalty.
- Combined Misses Per Instruction = I-Cache Miss Rate + LS Frequency × D-Cache Miss Rate
- Memory Stall Cycles Per Instruction = I-Cache Miss Rate × Miss Penalty + LS Frequency × D-Cache Miss Rate × Miss Penalty.
  Trong đó:
  - I-count: Tổng số lệnh.
  - LS-count: Số lệnh Load/Store.
  - LS Frequency: Tỉ lệ lệnh Load/Store trong chương trình.

Ví dụ 2: Cho chương trình có 106 lệnh, trong đó 30% lệnh loads/stores. D-cache miss rate là 5% và I-cache miss rate là 1%. Miss penalty là 100 chu kỳ. Tính combined misses per instruction and memory stall cycles.

- 1% + 30% * 5% = 0.025 combined misses. Mỗi lệnh tương đương 25 misses trên 1000 lệnh.
- Memory stall cycles = 0.025 * 100 (miss penalty) = 2.5 stalled cycles per instruction.
- Total memory stall cycles = $10^6 * 2.5 = 2,500,000$
- CPI Memory Stalls = CPI Perfect Cache + Mem Stalls per Instruction

Ví dụ 3: Cho CPI = 1.5 khi không có stall, Cache miss rate là 2% đối với instruction và 5% đối với data. Lệnh loads và stores chiếm 20%. Miss penalty là 100 chu kỳ đối với I-cache và D-cache. Tính CPI của hệ thống?

- Mem stalls cho mỗi lệnh = 0.02*100 + 20%*0.05*100 =3
- CPI Memory Stalls = 1.5 + 3 = 4.5 cycles

4.4 Thời gian truy xuất bộ nhớ trung bình

Average Memory Access Time (AMAT).
AMAT = Hit time + Miss rate × Miss Penalty
Do đó để giảm thời gian truy xuất:

- Ta giảm Hit time: bằng cách dùng bộ nhớ cache nhỏ (tăng miss rate :D), đơn giản.
- Giảm Miss Rate: bằng cách dùng bộ nhớ cache lớn , block size lớn (tăng Hit time) và k-way set associativity với k lớn.
- Giảm Miss Penalty bằng cách dùng cache nhiều mức.

Ví dụ 4: Tìm thời gian truy xuất trung bình khi biết hit time 1 chu kỳ, miss penalty 20 chu kỳ, miss rate 0.05 %. Biết máy tính chạy với tần số 0.5Ghz

- AMAT (cycles) = 1 + 0.05 × 20 = 2 cycles
- 0.5 Ghz → 1 chu kỳ 2ns
- AMAT (time) = 2×2 = 4ns

Ví dụ 5: Tìm thời gian truy xuất trung bình khi biết thời gian truy xuất cache L1 là 1 chu kỳ, thời gian truy xuấtcache L2 là 10 chu kỳ, thời gian truy xuất bộ nhớ chính là 100 chu kỳ, miss rate L1 5%, GLOBAL miss rate L2 10% (Global miss rate này bao gồm cả tỉ lệ miss của các bộ nhớ trước nó - trong trường hợp này là L1-cache). Biết máy tính chạy với tần số 1Ghz.

- AMAT (cycles) = 1 + 5% x (10 + Miss rate_L2%×100)
  = 1 + 5% x 10 + 5% x Miss rate_L2%×100
  = 1 + 5% x 10 + 10%×100 = 11.5 cycles
- 1Ghz → 1 chu kỳ 1ns
- AMAT (time) = 11.5×1= 11.5 ns

4.5 Bài tập

Bài 4.1. Cho bộ nhớ cache có dung lượng 256KB, block size là 4 word, mỗi lần truy xuất 1 byte. Xác định số bit của các trường tag, index, block offset trong các trường hợp.

- Direct mapped
- Fully associcative
- 2-Way set associcative

Bài 4.2. Cho bộ nhớ chính có dung lượng 1G, bộ nhớ cache có dung lượng 1MB, block size là 256B, mỗi lần truy xuất 1 word. Xác định số bit của các trường tag, index, block offset trong các trường hợp.

- Direct mapped
- Fully associcative
- 4-Way set associcative

Bài 4.3. Trong cache có 8 blocks, mỗi block là 4 words. Xác định số lần miss/hit khi hệ thống truy xuất vào các địa chỉ (dạng byte) theo thứ tự sau:

0x0001_002A
0x0001_0020
0x0002_006A
0x0002_0066
0x0002_0022
0x0001_002B

Trong các trường hợp:

- Direct mapped
- Fully associcative
- 2-Way set associcative

4.6 Đáp án/Gợi ý

Byte-offset: xác định byte trong block.
Word-offset: xác định word trong block.
index: xác định set trong block.
Tag: xác định block trong cache.
Tag + index: xác định block trong bộ nhớ chính.
Direct mapped: một block 1 set.
k-Way Set Associative: k block (k = 2^x) một set.
Fully associative: Tấc cả block 1 set.

Bài 4.1. Xác định tag, index, byte-offset.
Số phần tử trong 1 block = (size of block)/(size of phần tử truy xuất) = 4 word /1 byte = 4x4 bytes/ 1 byte = 16 = 2^4.
Số block trong cache = (size of cache) / (size of block) = 256 KB / 4 words = $2^8 * 2^{10}/4 * 4 = 2^{14}$ blocks.
Không gian địa chỉ là 32 bit.

- Direct mapped: byte-offset 4 bits, index = 14 bits, tag = 32 – 4 - 14 = 14 bits
- Fully associcative: byte-offset 4 bits, index = 0 bits, tag = 32 – 4 = 28 bits.
- 2-Ways set associative: 2 block tạo thành 1 set mà có 2^{14} blocks nên có 2^{13} sets, byte-offset 4 bits, index = 13 bits, tag = 32 -4 - 13 = 15 bits

Bài 4.2. Xác định tag, index, word-offset.
Số phần tử trong 1 block = (size of block)/(size of phần tử truy xuất) = 256B / 4 bytes = 2^6.
Số block trong cache = size of cache / size of block = 1MB / 256B = $2^{10}x2^{10}/2^8 = 2^{12}$ blocks.
Không gian địa chỉ là 1G, do đó ta dùng thanh ghi **30 bit** tính theo byte-offset.

- Direct mapped: word-offset 6 bits, index = 12 bits, tag = 30 – 6 – 12 – 2 = 10 bits.

- Fully associcative: word-offset 6 bits, index = 0 bits, tag = 30 – 6 – 2 = 22 bits.
- 4 ways set associative: 4 block tạo thành 1 set mà có 2^{12} blocks nên có 2^{10} sets, word-offset 6 bits, index = 10 bits, tag = 30 – 6 - 10 - 2 = 12 bits.

Chú ý: không gian tính theo byte thì phải chuyển về byte-offset để tính: số bit của địa chỉ = số bit của tag, index, và byte-offset.

Bài 4.3. HIT/MISS

Ta tính được có 4 bits byte-offset, 3 bits index. Do đó ta phân tích địa chỉ như bên dưới:
Cách 1:

Direct map

Address	Tag	Index	Offset	Miss/Hit	Giải thích
0x0001002A	0000 0000 0000 0001 0000 0000 0	010	1010	M	First access
0x00010020	0000 0000 0000 0001 0000 0000 0	010	0000	H	
0x0002006A	0000 0000 0000 0010 0000 0000 0	110	1010	M	First access
0x00020066	0000 0000 0000 0010 0000 0000 0	110	0110	H	
0x00020022	0000 0000 0000 0010 0000 0000 0	010	0010	M	Khác tag
0x0001002B	0000 0000 0000 0001 0000 0000 0	010	1011	M	Khác tag

Fully associative

Address	Tag	Offset	Miss/hit	Giải thích
0x0001002A	0000 0000 0000 0001 0000 0000 0010	1010	M	First access
0x00010020	0000 0000 0000 0001 0000 0000 0010	0000	H	
0x0002006A	0000 0000 0000 0010 0000 0000 0110	1010	M	First access
0x00020066	0000 0000 0000 0010 0000 0000 0110	0110	H	
0x00020022	0000 0000 0000 0010 0000 0000 0010	0010	M	First access
0x0001002B	0000 0000 0000 0001 0000 0000 0010	1011	H	

2 ways set associative, cần 2 bit index

Address	Tag	Index	Offset	Miss/hit	Giải thích
0x0001002A	0000 0000 0000 0001 0000 0000 00	10	1010	M	First access
0x00010020	0000 0000 0000 0001 0000 0000 00	10	0000	H	
0x0002006A	0000 0000 0000 0010 0000 0000 01	10	1010	M	First access
0x00020066	0000 0000 0000 0010 0000 0000 01	10	0110	H	
0x00020022	0000 0000 0000 0010 0000 0000 00	10	0010	M	Khác tag
0x0001002B	0000 0000 0000 0001 0000 0000 00	10	1011	M	Khác tag

Cách 2:

- Lấy địa chỉ chia cho kích thước của block được kết quả (A) dùng để xác định block trong RAM
- Lấy kết quả (A) modulo số set được kết quả là index.
- Lấy kết quả (A) chia số set được kết quả là tag

8 blocks cache → mỗi block là 4 words.

Direct map 8 sets và mỗi block 16 bytes.

Address	Address/block size = A	Tag = A / 8	Index = A % 8	Miss/Hit	Giải thích
0x0001002A	4098	**512**	**2**	M	First access
0x00010020	4098	512	2	H	
0x0002006A	8198	**1024**	**6**	M	First access
0x00020066	8198	1024	6	H	
0x00020022	8194	**1024**	**2**	M	Khác tag
0x0001002B	4098	**512**	**2**	M	Khác tag

Fully associative

Address	Address/block size = A	Tag = A	Miss/hit	Giải thích
0x0001002A	4098	**4098**	M	First access
0x00010020	4098	4098	H	
0x0002006A	8198	**8198**	M	First access
0x00020066	8198	8198	H	
0x00020022	8194	**8194**	M	First access
0x0001002B	4098	4098	H	

2-way set associative, cần 2 bit index

Address	Address/block size = A	Tag = A / 4	Index = A % 4	Miss/Hit	Giải thích
0x0001002A	4098	**1024**	**2**	M	First access
0x00010020	4098	1024	2	H	
0x0002006A	8198	**2049**	**2**	M	First access
0x00020066	8198	2049	2	H	
0x00020022	8194	**2048**	**2**	M	Khác tag
0x0001002B	4098	**1024**	**2**	M	Khác tag